Overview

- *Administrative*
  - HW 2 Grades
  - HW 3 Due
- *Topics:*
  - What are Threads?
  - Motivating Example : Async. Read()
  - POSIX Threads
  - Basic Thread Management
  - User vs. Kernel Threads
  - Thread Attributes
- *Readings: Chapter 9 (pp. 333-364)*
- *Exercises: 9.1 - 9.3*

What are Threads?

- *Thread of execution in a program:*
  * Flow of Control for a process
  * Sequence of instruction executed by CPU for a process
- *Ex 9.1: (pp. 333)*
  * process A executes statements a5, a6, a7 in a loop
  * process B executes statements b2, b3, b4, b5 in a loop
  * P1 sees 1 thread: a5, a6, a7, a5, a6, a7, ...
  * P2 sees 1 thread: b2, b3, b4, b5, b2, b3, b4, b5, ...
  * CPU and OS see interleaved threads from P1 and P2,
  - e.g. a5, a6, b2, b3, b4, b5, b2, a7, a5, b6, ...
- *Q? Why not a user process w/ multiple threads ?*
  * Multiple blocking I/O channels (e.g. sockets)
  * Responsive user interfaces
  * Server program handling concurrent requests
  * Simplify writing parallel programs
  * Programs using multi-processor machines

Hello World with 2 Processes

- *Example with Processes (w/o synchronization)*

```
void print_message_function( void *ptr );
main()
{
  pid_t process1, process2;
  char *message1 = "Hello";
  char *message2 = "World";

  if ( (process1 = fork()) == 0) {
        print_message_function( message1 );
                exit(0);
  } if ( (process2 = fork()) == 0) {
        print_message_function( message2 );
                exit(0);
  }
  /* wait() for children to finish */
}

void print_message_function( void *ptr )
{
  char *message;
  message = (char *) ptr;
  printf("%s ", message);
}
```

Hello World with 2 Threads

- *Example with Threads (w/o synchronization)*

```
void print_message_function( void *ptr );
main()
{
   pthread_t thread1, thread2;
   char *message1 = "Hello";
   char *message2 = "World";

   pthread_create( &thread1, pthread_attr_default,
       (void*)&print_message_function, (void*) message1);
   pthread_create(&thread2, pthread_attr_default,
       (void*)&print_message_function, (void*) message2);

   exit(0);
}

void print_message_function( void *ptr )
{
   char *message;
   message = (char *) ptr;
   printf("%s ", message);
}
```

Threads vs. Processes

- *Concurrent Program architecture*
  * Cooperating group of processes
  * Group of threads within a process
  * Mixed
- *Processes as units of concurrent execution*
  * +Security: a buggy process won't affect other processes
  - Example: unix shell
  * +Pipes: Simple synchronization
  * - Slow Shared Synchronization variables (Ch. 8)
  * - High costs: memory, creation, context switch...
  * - Severe limits on number of processes (concurrency)
- *Threads*
  * Share code and data across all threads
  * Reduce context switches overheads
  * Faster creation, synchronization: Table 9.2 (pp. 360)
  * - Shared memory => race conditions
  * - Weak security boundaries

9.1 A Long Example

- *Problem: Monitoring multiple file descriptors*
  - No order on arrival of input across channels
  - Non-blocking read()
- *Alternative Solutions:*
  - 9.1.1 non-blocking read() with polling
  - 9.1.2 asynchronous I/O with signal
  - 9.1.3 ’select’ statement
  - 9.1.4 system call ’poll()’
  - 9.1.5 Threads

9.1.5 Monitoring I/O channels

- *poll_and_process(int fd)*
  * Program 9.1 (pp. 36)
  * Called for each file descriptor
  * By most solutions
  * Error handling is complex
  - -1 => error, no input, signal
  - check errno for EINTR, EAGAIN
- *Simple solution: non-blocking I/O*
  * Program 9.2 (pp. 336-7)
  * Get filenames from command line
  * Open two file descriptors w/ O_NONBLOCK
  * While loop to poll file descriptors
- *Comments* - *Busy waiting*
  * - Single thread of control
  * - Long request delays other requests

9.1.5 Monitoring I/O channels w/ Signals

- *Signal based solution - no busy wait*

  * Program 9.3 (pp. 348-350)

  * Use SIGPOLL signal to communicate b/w
  - device driver and the main() program

  * SIGPOLL blocked except during sigsuspend()

- *Strategy*

  * Open file descriptors for non-blocking I/O

  * Block SIGPOLL signal

  * Install signal handler for SIGPOLL

  * Signal handler flags arival of SIGPOLL
  - via a global variable
  - Recall Example 5.20 (pp. 184, sigsuspend())

  * Ask device driver to send SIGPOLL signal
  - ioctl() with I_SETSIG flag

  * Loop on { polling and sigsuspend() }

- *Comments:* - *Complex logic*

  * - Single thread of control

  * - Long request delays other requests

9.1.5 Monitoring I/O channels w/ Threads

- *Thread based solution - Program 9.7 (pp. 347)*

  * monitor_fd(fd_array[], num_fd)

  * multiple threads

  * Assign a file descriptor to each thread
  - function process_fd()

  * Ensure no conflict in FDT, file descriptors, ...

  * Wait for threads to finish

- *Program 9.6 - Details of process_fd()*

  * Get file descriptor as argument

  * infinite loop over
  - blocking read from file descriptor
  - process command

- *Comments: - Simple logic*

  * - Long request don't delay other requests

  * - No busy wait

Threads vs. Procedures

- *Both share global variables and heap*
- *Procedures without threads*
  - * Decompose source code into procedures
  - * Example Program 9.3 (pp. 344)
  - * Single Thread of control: Figure 9.1 (pp. 345)
  - * Single stack of activation records
  - * A blocking I/O in a procedure
  - - may halt entire process
- *Threads*
  - * Each thread executes a procedure
  - * Example Program 9.4 (pp. 346)
  - * Multiple threads active - Figure 9.2 (pp. 345)
  - * A blocking I/O in a thread
  - - does not halt entire process
  - * Program 9.7 (pp. 346-7)
  - - Note: "process_fd()" uses blocking read

9.2 POSIX Thread Abstract Data Type

- *Abstract Data Type = <Attributes, Operations>*
  - * Examine Fig. 2.1 (pp. 32) and identify
  - \- What's unique to a thread of execution in a process?
  - * execution stack, register set, PC, state
  - * Share- code, heap, global data, environment, pid, ...
- *Attributes*
  - * Stack size
  - * Stack Address
  - * Scope
  - * Schedule Policy
  - * Schedule Parameters, e.g. thread priority
- *Operations : See Table 9.3 (pp. 360)*
  - * Initialization
  - * Detach State
  - * Inherit Schedule
  - * Get/Set Attributes

9.4 User vs. Kernel Threads

- *Thread Implementations*

  * OS Kernel level

  * User level

- *User Level Threads*

  * Threads within a process

  * Compete among each other for process resouces

  * Scheduled by a run-time library linked to process code

  * A blocking system call by a thread can block other threads,

  *   So these calls may be postponed

  * + Low overhead

  * - Has limited resources

  * - Run-time library must get control periodiclly for scheduling

  *   --> complex code for threads

9.4 User vs. Kernel Threads

- *Kernel Level Threads*
  - Threads are visible to OS Kernel
  - Threads complete for system wide resources
  - Can take advantage of multiple processors
  - More expensive than user level threads
  - Scheduling can be as costly as process scheduling
  - See Table 9.2 (pp. 360) for comparison!
- *Hybrid Model: (Fig 9.5, pp. 359)*
  - User writes programs interms of user level threads
  - And specified number of kernel-level threads
  - User level threads are mapped to kernel level threads

9.3 POSIX Threads: system calls

- *Thread Package Has*
  - * A runtime library to manage thread ADTs
  - * In a user transparent manner
  - * Has calls to create, delete, synchronize
  - * Calls return 0 if and only if successful
  - * Table 9.1 (pp. 348) illustrates two pakages
- *Support dynamic threads*
  - * Can be created at any time during execution
  - * Number of threads not specificed in advance

9.3 POSIX Threads: system calls

- *pthread_create()*
  - * Create a thread to execute given function
  - * Example 9.4 (pp. 346)
  - * Synopsis: pp. 349
  - * Parameter1: thread id
  - * Parameter2: thread attribute object
    - \- NULL => default values
  - * Parameter3: function to be executed by thread
    - \- retriction: 1 argument (* void), returns (* void)
    - \- retriction similar to signal handler
  - * Parameter4: the argument to the function
  - * Returns: error code

9.3 POSIX Threads: system calls

- *Simulate procedure-call synchronization*
  - * pthread_exit() - pthread_join() pair
  - * Can exchange data between threads!
  - * Recall process system calls
  - - exit(status) - wait() synchronization
- *pthread_exit()*
  - * Terminate the calling thread
  - * Takes an agument (void *)
  - - for return value via pthread_join()
- *pthread_join()*
  - * Wait for specific child thread
  - * Arguments1: thread id to wait on
  - * Arguments2: result from thread waited on
  - - e.g. "errno" my be returned by thread

9.3 POSIX Threads: system calls

- *Example: Copying multiple files*
  - * Program 9.9 (pp. 351-2)
  - * Exercise 9.1 , 9.2(pp. 353)
  - * Exercise 9.3 (pp. 355)
- *pthread_self()*
  - * Find your own thread_id
- *Synchronization Issues (Chapter 10)*
  - * Changing values of shared data, e.g. reference parameter
  - * System calls should be thread-safe (i.e. no thread-switching)

9.5 Thread Attributes

- *Recall Thread Attributes*
  * Stack size
  * Stack Address
  * Scope
  * Schedule Policy
  * Schedule Parameters, e.g. thread priority
- *Reading/Writing attributes*
  * Example: priority of a thread
  * Example 9.6 (pp. 362)